Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Using the meta-environment for domain specific language engineering

T. van der Storm, J.J. Vinju

REPORT SEN-R0805 OCTOBER 2008

Centrum Wiskunde & Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Using the meta-environment for domain specific language engineering

ABSTRACT
The development of a domain specific language (DSL) can be a difficult and costly undertaking. Language workbenches aim to provide integrated development support to ease this process. The Meta-Environment is a language workbench providing parsing, analysis, transformation, syntax highlighting and formatting support for the development of programming languages. In this paper we elaborate on the suitability of it for DSL engineering by reporting on our experience in developing a little language for markup generation.

1998 ACM Computing Classification System: D.2.6; D.3.4
Keywords and Phrases: Domain Specific Languages; Language Workbenches; Markup Generation

Using The Meta-Environment for Domain Specific Language Engineering

Tijs van der Storm
CWI
Kruislaan 413, 1098 SJ
Amsterdam, The Netherlands
storm@cwi.nl

Jurgen Vinju
CWI
Kruislaan 413, 1098 SJ
Amsterdam, The Netherlands
jurgenv@cwi.nl

Abstract

The development of a domain specific language (DSL) can be a difficult and costly undertaking. Language workbenches aim to provide integrated development support to ease this process. The Meta-Environment is a language workbench providing parsing, analysis, transformation, syntax highlighting and formatting support for the development of programming languages. In this paper we elaborate on the suitability of it for DSL engineering by reporting on our experience in developing a little language for markup generation.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**] Programming Environments—*Integrated & Interactive Environments*
D.3.4 [**Programming Languages**] Processors—*code generation, interpreters, parsing*

General Terms

Languages, Design.

Keywords

Domain Specific Languages, Language Workbenches, Markup Generation

1 Introduction

Domain Specific Language (DSL) development can be a daunting task. Whereas tool support for DSLs traditionally has concentrated on syntax definition and code generation, language workbenches aim to provide *integrated* development support. A question remains, however, whether these "IDEs for languages" live up to their promise. In this paper we report on our experience of developing a DSL for markup generation in one such language workbench: the ASF+SDF Meta-Environment [7, 11].

Recently, Martin Fowler has called "language workbenches" the "killer apps for domain specific languages" [4]. In that article Fowler describes his experience of building a simple DSL using the JetBrains Meta-Programming System [3]. JetBrains MPS is just one of many language workbenches, some of which are developed in the context of research. Moreover, some of these have been around for a long time. For instance, Heering et al. mention at least 21 language development systems and tool kits that have been used for DSL development [9].

More recently a number of projects explicitly targeting IDE support for language development have gained some attention. ANTLRWorks is an IDE built around the ANTLR parser generator by Terence Parr [10]. This environment allows editing, testing and debugging of ANTLR language definitions using advanced visual features.

Another approach to IDE support for language development is the Spoofax environment, which is built on top of Eclipse [5]. Within this environment the language developer can edit SDF syntax definitions and Stratego transformation specifications [1].

Building on top of an existing IDE such as Eclipse is attractive since it presents the opportunity to reuse components, such as outline, navigation, error reporting views and the integration with version con-

trol. In the context of Eclipse there is one project which focuses in particular on providing not only IDE support for language development (meta level), but also for the language implementations themselves (object level). The Eclipse IDE Meta-tooling Platform (IMP) aims to provide a platform for high-quality language-specific IDEs [2]. SDF2IMP [6] is a collection of tools that bridge SDF syntax definitions to IMP and offers other declarative configuration languages to interface with the IDE features of IMP.

In this paper we describe our experience of developing a small DSL for generating (X)HTML markup, WÆBRIC. This language is complex enough to exercise many aspects of the ASF+SDF Meta-Environment. The Meta-Environment has been used before for DSL development in the context of financial products [13]. It was used to prototype the parser, type checker and compiler for a DSL called RISLA. However, the Meta-Environment has seen substantial changes since then. In fact, the current version is the third incarnation, featuring a modern GUI, automatic syntax highlighting, extensible error reporting and data visualization.

2 The Meta-Environment

The Meta-Environment[1] is organized around two formalisms: the Syntax Definition Formalism (SDF) and the Algebraic Specification Formalism (ASF). We briefly describe both formalisms.

2.1 All Data Has Syntax

SDF allows the modular specification of arbitrary context-free grammars. In the Meta-Environment it is used to define the presentation of *all* data, not just the syntax of programming languages. For instance, when specifying an interpreter, one often needs an environment structure to hold values that have been bound to variables. Such data structures are also defined using context-free grammars. The ASF+SDF standard library provides numerous "standard" grammars for common data structures, such as list, sets, integers, strings, identifiers etc. Additionally, it contains definitions for a number of environment specific data formats: errors, pretty printer expressions, source locations, relation stores, and parse trees.

[1] http://www.meta-environment.org

2.2 Computation is Transformation

Whereas all data representations are defined in SDF, all computation is expressed in ASF. The units of computation in ASF are conditional rewrite rules, called "equations". An ASF equation consists of a left-hand side and a right-hand side, and zero or more condition clauses. Left-hand sides and all condition parameters are patterns over arbitrary language constructs - in concrete syntax - defined in SDF. We will provide more details of ASF as we go.

2.3 Integration is Transformation

The Meta-Environment IDE contains a number of programmable features such as editors with syntax highlighting, error panes, project browsers, fact browsers and graph and chart visualizations. Some features infer language properties directly from (annotated) SDF syntax definitions, such as the syntax highlighter, other features require some transformation code.

The general principle is that each IDE feature has a data or configuration language as its interface. After the language engineer provides a transformation from a DSL to this language, the Meta-Environment automatically integrates the transformation into the IDE. The transformation is scheduled when appropriate and the IDE will display the result appropriately.

For example, when we provide a translation from WÆBRIC to the "error language", the error pane will be filled with static WÆBRIC errors while editing a program, including clickable source locations. Another example, to visualize the import structure of WÆBRIC programs, the user may translate each WÆBRIC program to records in the "relation store" language for fact representation. As a result, the IDE user will be able to see an import graph of a WÆBRIC program.

3 Case-study: Markup Generation

WÆBRIC is a small language for generating XHTML markup. Its design is motivated by the lack of programmer friendly abstraction facilities in existing markup languages. Typing markup is not for humans and WYSIWYG editors have their own issues like generating totally inaccessible and unmaintainable XHTML code. WÆBRIC instead, provides a user-friendly syntax to factor Web pages in self-contained functional building-blocks. We

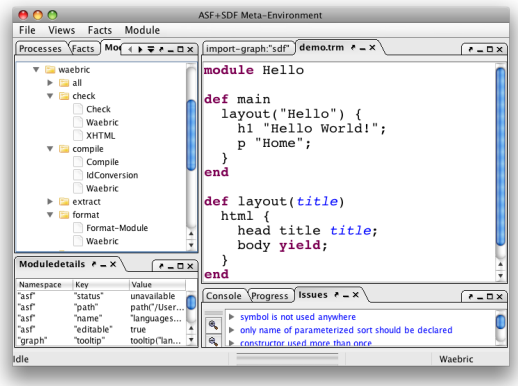

Figure 1: Screen shot of the ASF+SDF Meta-Enviroment displaying a snippet of a WÆBRIC program

choose not to provide related work in the area of markup generation since that is incompatible with the object of this paper.

The basic WÆBRIC program consists of a number function definitions, possibly partitioned over a number of modules, together with a site definition which maps WÆBRIC markup expressions to XHTML files. A simple WÆBRIC example is displayed below:

```
def main
    layout(”Hello”) {
        h1 ”Hello World!”;
        p ”Home”;
    }
end

def layout(title)
    html { head title title; body yield; }
end
```

Running this program produces a single XHTML document which consists of the markup generated by the function main. The main function calls another function, called layout, which receives an string argument “Hello”. Additionally, a block (enclosed in curly braces) is passed to layout. This block will be evaluated wherever the statement **yield** occurs within the implementation body of layout. The block itself consists of two “function calls”, to h1 and p. Since no definition exists for h1, this will produce a XHTML “h1” element with the string “Hello World!” as content. Subsequently a paragraph is rendered containing the string “Home”.

The layout function defines a skeleton page framework that can be reused across different pages. In the example it sets up a basic XHTML document with header and title; the body of a page is obtained from the block passed into it. Note how nesting of elements along a single spine in the document tree requires no curly braces (as can be seen from how the title element is included in the head element).

Running this example will produce the following (XHTML) markup:

```
<html> <head> <title> Hello </title>
</head> <body> <h1>Hello World!</h1>
<p>Home</p> </body> </html>
```

In addition to the basic markup generating function calls, as illustrated by the example, WÆBRIC contains an if-then-else statement, an each statement, for iterating over sequences of values, a let construct for the introduction of local variables and local redefinition of functions and string interpolation syntax (for embedding markup directly in text).

The example in Figure 2 shows how recursive menus could be defined in WÆBRIC. The first function, menu, receives a data object (*menu*) containing the labels, URLs and sub-menus that should be rendered in XHTML. The next statement just renders the title of the (current) menu using the built in statement echo. After the title follows an unordered list containing the items of this menu. For each element in the “kids” property of *menu* the menu-item function is called.

The menu-item function first checks whether this *item* has any children (sub-menus). If so, it produces a li(st) element containing the output of a recursive call to menu. If there are no sub-menus, the result is a list element with an anchor tag which links the title of item to its URL. The result of an invocation (with the appropriate data for the *menu* parameter) of menu could look like the screen shot next to the source code.

```
def menu(menu)
    echo menu.title;
    ul {
      each (kid: menu.kids)
          menu-item(kid); }
end

def menu-item(item)
    if (item.kids) li menu(item);
    else li a(href=item.link) item.title;
end
```

Menu

- Home
- Misc
  - Contact
  - Links

Figure 2: Recursive menus in WÆBRIC and the possible output

4 Implementation of WÆBRIC

4.1 Syntax

The syntax of WÆBRIC is defined in SDF. The grammar rules are divided over 12 modules according to non-terminal symbol. For instance, the syntax for Statements is defined in a single module, for Expressions in another, and so on. A snippet of the Statement grammar is displayed in Figure 1. The example shows the syntax of the if-then-else statement, block-statements, the each iteration construct, and the primitive echo statement. Note that the direction of the productions is reversed, compared to (E)BNF.

Since SDF supports the definition of arbitrary context-free grammar there is a risk of writing ambiguous grammars. SDF provides disambiguation rules to amend such situations. The example contains an advanced disambiguation construct. The last line of Figure 1 defines that an if-statement without an else may not immediately be followed by the keyword "else". The definition uses an empty non-terminal (NoElseMayFollow) to indicate the position in the grammar where this rule needs to be applied. This declaratively prevents the "dangling else" ambiguity for WÆBRIC. We come back to the topic of disambiguation in Section 6.

context-free syntax

"if" "(" Predicate ")" Stm NoElseMayFollow	→	Stm
"if" "(" Predicate ")" Stm "else" Stm	→	Stm
"{" Stm* "}"	→	Stm
"each" "(" Var ":" Expression ")" Stm	→	Stm
"let" Assignments "in" Stm* "end"	→	Stm
"echo" Expression ";"	→	Stm

context-free syntax

→ NoElseMayFollow

context-free restrictions

NoElseMayFollow -/- [e].[l].[s].[e]

Table 1: Statement syntax of WÆBRIC in SDF

4.2 Import Resolving

Parsers derived from SDF grammars only parse one file at the time. This poses a problem for modular DSLs that involve a multiplicity of files. There are basically two approaches to circumvent this limitation:

Preprocessing the source code Combine all modules in one file by textual concatenation. Use, for instance, regular expressions in some general purpose language to find all imported modules.

Loading and parsing from within ASF It is currently possible to load and parse files in ASF using I/O functions from the standard library.

The first option has two disadvantages: the regular expressions repeat knowledge from the syntax definition and it does not integrate directly with the Meta-Environment. Since the Meta-Environment is actively used for debugging, testing and IDE prototyping this would seriously hamper the development process. We therefore have chosen for the second approach.

The process of resolving imports starts with a top module. For this module the set of imported modules is determined. These are then loaded, parsed, and added to the list of current modules. Then for each of the new modules, the process repeats itself, until no new modules get added.

4.3 Well-formedness Checking

The Wæbric implementation contains a well-formedness checker developed in ASF. When run, this tool returns a (possibly empty) list of "errors". The syntax of Errors is defined in the standard library of ASF+SDF and forms the interface to the error pane in the GUI of the Meta-Environment. This way, user defined checkers are seamlessly integrated,—an instance of "integration is transformation". When an error is detected, it will be visually presented to the user. Furthermore, each error is linked to its source origin so that when the user clicks on an error, the cursor jumps to the location of the offending construct.

The checker is implemented in two phases. The first phase consists of extracting all relevant facts from a WÆBRIC program. These facts are represented as relations (also defined in the standard library of ASF+SDF) between relevant source code artifacts. An example of such a fact could be: the arity of function "foo" is three. In relational form this fact could be represented as a tuple $\langle$"foo", 3$\rangle$ in a relation *Arity*. Other relations that could be extracted might include: call graphs, use-define relations, import relations etc. Such relations can be considered semantic projections on source code [8].

In the implementation of WÆBRIC, the extracted relations are used by the checker. Using relational calculus the extracted relations can be queried, combined and enriched in arbitrary ways. Thus, for instance, calls to functions that provide the wrong number of arguments can be determined as follows. Given the relation $Arity \subseteq String \times \mathbb{N}$ and a relation

```
[if1] eval-pred($pred, $env) == true
      ======================================
      eval-stat(if ($pred) $stat1 else $stat2,
                $env, $block, $module*) =
        eval-stat($stat1, $env, $block, $module*)

[if2] eval-pred($pred, $env) == false
      ======================================
      eval-stat(if ($pred) $stat1 else $stat2,
                $env, $block, $module*) =
        eval-stat($stat2, $env, $block, $module*)
```

Figure 3: Evaluation rules for WÆBRIC if-statements. Keywords are in bold-face, variables are italicized and start with "$".

$Calls \subseteq Location \times (String \times \mathbb{N})$, where *Location* is a data type for source locations. The set of incorrect calls are then computed with the following expression: $\{\langle loc, \langle fun, n\rangle\rangle \in Calls \mid \langle fun, n\rangle \notin Arity\}$.

4.4 Interpreter & Compiler

WÆBRIC programs can be evaluated in two ways: using an interpreter, for testing and debugging from within the Meta-Environment, and using a compiler which generates a Java class which can be deployed in a web server. Below we briefly review how these two essential components are implemented.

The interpreter directly implements the operational semantics of WÆBRIC programs. The evaluation functions receive a WÆBRIC program, and environment, the current block, and the total set of modules as arguments. Evaluation starts at the "main" function. The result is an XHTML document.

The rules for evaluating if-statements is displayed in Figure 3. On the left-hand side of the equals sign, the "eval-stat" function matches a (concrete) program pattern, namely an if-statement. On the right-hand side the result of evaluating the if-statement is returned. The result, obviously, depends on the result of the condition. An auxiliary function, eval-pred, returns true if a conditional expression is valid in the current environment.

In order to allow WÆBRIC markup programs to be used in dynamic contexts (such as, for instance, Java Servlets), we designed a compiler to Java. This compiler is a simple transformation of WÆBRIC function definitions to Java methods, wrapped in a single Java class. Figure 4 lists three rewrite rules that compile a WÆBRIC statement to Java statements. Note how the left-hand sides are using WÆBRIC syntax, while the right-hand sides are using Java syntax. For example, the WÆBRIC echo statement corresponds to a Java statement that uses an OutputWriter object to write a string value.

```
[0]   stat-to-jstats({ &stats }) =
        stats-to-jstats(&stats)

[1]   stat-to-jstats(echo &exp;) =
        $out.write(exp-to-jstring(&exp));

[2]   stat-to-jstats(if (&exp) &stat) =
            if (exp-to-jexp(&exp) != null) {
                stat-to-jstats(&stat)
            }
```

Figure 4: Rewrite rule defining the transformation of some WÆBRIC statements to a Java statements. Meta-variables start with "&". The *$out* (Java) variable is a parameter to the method that will contain these statements.

4.5 Wæbric and XML Formatters

Next to the essential components covering parsing and evaluation, we developed formatters for WÆBRIC programs and XHTML documents. Formatters are defined in ASF+SDF by writing a transformation to a layout language, called Box [12]. We do not describe these transformations in more detail. Suffice it to say they are (almost) isomorphic mappings of WÆBRIC/XHTML constructs to Box expressions. The Meta-Environment then evaluates these Box expressions and produces the pretty printed source code.

5 Results

The sizes of the WÆBRIC components are listed in Table 2 in SLOC. In total, around 5000 SLOC have been required to develop this complete tool chain. All in all, it took the first author 2 weeks to develop all components, with the exclusion of the Java 1.5 and XML grammars, which could be reused. Note that more SLOC are spent in SDF. This can be explained from the fact that "all data has syntax", even function signatures and pattern variables used in ASF equations.

The current implementation of WÆBRIC has been used in the generation of a small static website[2]. Since this site originally was developed using a WYSIWYG XHTML editor, its pages contained a lot of duplication. Each page for instance, contained the same menu and used the same customized headers. Using WÆBRIC the first author was able to factor out such commonality in function definitions that were reused in every page definition.

The description of the site in WÆBRIC caused a

[2] http://www.lavaliterair.nl

Component	SDF	ASF	Total SLOC
Java 1.5 grammar	1583	0	1583
XML grammar (w/o DTDs)	109	0	109
WÆBRIC grammar	248	0	248
Import resolver	106	87	193
Fact extractor	84	119	203
Checker	131	174	305
Evaluator	348	685	1033
Compiler	277	611	888
WÆBRIC formatter	129	43	172
XHTML formatter	50	100	150
Total:	3065	1819	4884

Table 2: Size of WÆBRIC components in Source Lines of Code (SLOC). The Java and XML grammars have been reused and have not been developed specifically for WÆBRIC.

reduction in size of approximately 40%, measured in (non-empty) source lines of code (SLOC). To measure this in the generated XHTML, the markup was pretty-printed first. Whatever the specific meaning of such figures may be, in our experience the pay-off was immediate: change requests for the site now only involved a single point of change.

6 Evaluation

In this section we reflect on the suitability of the Meta-Environment IDE and the ASF+SDF formalisms for DSL engineering. Do they have all the necessary features? Most importantly, since the Meta-Environment is a programmer's workbench, is the behavior of these features tractable and debuggable?

ASF+SDF focuses on simplicity, modularity and declarativeness. Design elements such as "all data is syntax" and "computation is transformation" make that possible. On the one hand these features supported development of WÆBRIC directly, since it is mostly about syntax and transformation. ASF+SDF gave us a head start by automatically providing parsers, pattern matching, tree constructions, tree traversal and source-to-source transformation.

On the other hand, we observe that DSLs are not only about implementing a parser and transformations. Additional data-structures are needed, such as sets to store WÆBRIC function definitions. Additional data-flow is needed to thread this set to its use sites. These features are easily simulated using ASF+SDF, but with some disadvantages. The extra implementation layer diminishes tractability, efficiency and conciseness. Therefore we propose to include sets and relations as primitive data-types for ASF+SDF and to include parametrized function definitions and global or dynamically scoped variables in future versions of ASF+SDF.

The SDF formalism offers highly declarative syntax definition. This left us to focus on the actual syntax of the language, conveniently ignoring technical details such as non-determinism, grammar factoring and other parsing boilerplate. We do have a note on the tractability of SDF: it cannot report if a grammar is ambiguous or not. SDF reports ambiguity only after parsing ambiguous WÆBRIC programs. This introduces uncertainty about the quality of the grammar. We had to convince ourselves by studying our grammar, including its non-trivial disambiguation rules, that it is not ambiguous anymore. Therefore we propose to add conservative grammar analyses to SDF that can state with certainty that a grammar is not ambiguous.

Meta-Environment. On the one hand the Meta-Environment has the basic features that one expects from any IDE and advanced features specifically targeted at language engineering. It includes syntax directed editing and syntax highlighting for ASF+SDF itself as well as for the object language, WÆBRIC, automatically. It includes static checkers for ASF+SDF that help debugging the WÆBRIC implementation. It includes debugging and inspection facilities, such as stepwise debugging of rewriting steps and parse tree visualization. It includes libraries of reusable syntax definitions and data-structures, such as the XML syntax definition. Most importantly, the Meta-Environment allowed us to modularly and iteratively implement parts of WÆBRIC and immediately integrate them into the IDE for on-line testing and debugging.

One the other hand the Meta-Environment misses a number of standard IDE features, such as integration with version control, outline and auto-complete. It's incremental build feature is sometimes slow while regenerating parsers for the highly modular implementation of WÆBRIC. We therefore propose to build the Meta-Environment on top of the Eclipse platform and to redesign its incremental build feature for efficiency.

6.1 Discussion

It is instructive to reflect for a moment on how our implementation of WÆBRIC using the Meta-Environment relates to alternatives.

Most other parser generation frameworks require a language's grammar to be moulded to fit in specific subclasses of all context-free grammars, such

as LR(1). As indicated above, SDF enabled us to concentrate much more on the syntax of the language itself instead of dealing with parser details.

Implementing the interpreter in a general purpose language, such as, for instance, Java in the case of ANTLR [10], could arguably be easier because of the availability of many more data structures and libraries. Other advantages would include better integration with the OS (for instance, for dealing with files) and better performance.

Intuitively, compilers *are* transformations, and consequently this is where ASF+SDF shines. Declarative transformation rules that guarantee syntactic correctness have distinct advantages over the use of `printf` statements or textual template languages.

7 Conclusion

We have described our experience using the ASF+SDF Meta-Environment for engineering WÆBRIC, a little language for XHTML markup generation. It is a non-trivial DSL featuring function abstraction, conditionals, iteration, local rebinding of variables and functions, recursion and block closures. This language has been implemented using ASF+SDF. The tool set supporting WÆBRIC, consists of a well-formedness checker, an interpreter, a compiler to Java, and two source formatters (WÆBRIC and XHTML). These tools where integrated into The Meta-Environment IDE.

Based on our experience we have elaborated upon the suitability ASF+SDF Meta-Environment for engineering DSLs and formulated possible improvements. We assessed the Meta-Environment on two accounts: IDE features and formalism features. All and all we conclude that the ASF+SDF Meta-Environment is quite suitable for DSL engineering. There are some rough edges that can be taken care of by changing both ASF+SDF to improve its suitability for DSL features that are not directly serviced by parsing or rewriting.

References

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

[2] P. Charles, R. M. Fuhrer, and S. M. Sutton Jr. IMP: a meta-tooling platform for creating language-specific IDEs in eclipse. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 485–488. ACM, 2007.

[3] S. Dmitriev. Language oriented programming: The next programming paradigm. Online: http://www.onboard.jetbrains.com/is1/articles/-04/10/lop/mps.pdf, 2004.

[4] M. Fowler. Language workbenches: The killer-app for domain specific languages? Online: http://martinfowler.com/articles/language-Workbench.html, June 2005.

[5] K. T. Kalleberg and E. Visser. Spoofax: An interactive development environment for program transformation with Stratego/XT. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA07)*, pages 47–50, Braga, Portugal, March 2007.

[6] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, Budapest, Hungary, April 2008.

[7] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[8] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[9] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[10] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[11] M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE Computer Society Press, 2007.

[12] M. van den Brand, A. Kooiker, J. Vinju, and N. Veerman. A Language Independent Framework for Context-sensitive Formatting. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society Press.

[13] A. van Deursen and P. Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, 10:75–92, 1998.